Modeling Context-Aware Distributed Event-Based Systems

Eduardo S. Barrenechea, Rolando Blanco, and Paulo Alencar
David R. Cheriton School of Computer Science
University of Waterloo

Technical Report CS-2010-07

Abstract

Emerging applications are becoming increasingly dynamic, adaptive and context-aware in areas such as just-in-time location-based m-commerce, situational health monitoring, and dynamic social networking collaboration. Although numerous systems and implementation infrastructures have been proposed to deal with some features of such systems, there is a lack of higher-level modeling abstractions and associated component metamodels that combine dynamic features, adaptation mechanisms, and context-awareness.

In this report we propose a metamodel for context-aware distributed event-based systems that supports novel forms of event-based context-aware interactions such as context event schemas, component interfaces that react to context, event and context interactions at the interface level and subscriptions based on both events and context. Our approach also supports requirements for context-aware systems proposed in the literature. The applicability of the approach is illustrated by showing how existing context-aware systems can be modeled using our metamodel.

1 Introduction

Recent emerging applications are becoming increasingly dynamic, adaptive and context-aware in areas such as just-in-time location-based m-commerce, situational health monitoring, and dynamic social networking collaboration. Context-awareness provides systems with increased information about the situations in which its components and users are immersed. Context is highly dynamic, sometimes with drastic changes. For example, a change in location can have an effect on the status of the available services or behaviour of running applications. Context-aware systems are required to support this high level of dynamism.

Although numerous systems and implementation infrastructures have been proposed to deal with features of context-aware systems in isolation, there is a lack of component metamodels and higher-level modeling abstractions. Currently, context-awareness representation is typically addressed at the application level. The coupling between application and context-awareness related code introduces problems associated with software architecture and development, reusability, maintenance and program comprehension. By introducing context and context-awareness entities as first class citizens, separate from application level entities, it is possible to model context-awareness in an application-independent manner. This promotes reusability of concepts and features, as well as strengthens support for program comprehension, maintenance and architecture.

The features related with context-awareness require a highly dynamic infrastructure. Distributed event-based systems (DEBSs) provide such infrastructure through adaptive component interfaces and loosely coupled components communicating via event-based asynchronous interactions. The low coupling between components in a DEBS is due to the fact that components generating events are oblivious to components consuming the events and vice-versa. Consumers are interested in event types and not in the components generating events. This property of DEBSs allows context to be disseminated in a transparent fashion throughout the system. Context-aware components are notified of context related events generated from changes in context irrespectively of context sources. Context sources and context-aware components can enter and leave the system without the need to alter existing components.

In this report we propose a metamodel for context-aware distributed event-based systems. The metamodel supports the dynamism associated with context-awareness through event-based interactions and component interface changes. The metamodel consists of a structural view and a control view. The structural view is used to model the relationships be-

tween events and interfaces. The control view is used to model administrative components in a DEBS, as well as component grouping and access controls that can be imposed on events and interfaces. The model meets general requirements for context-aware systems proposed in the literature and later discussed in this report. The metamodel supports novel forms of event-based context-aware interactions, such as context event schemas, component interfaces that react to context, event and context interactions at the interface level, and subscription based on both events and context.

1.1 Distributed Event-Based Systems

A distributed event-based system (DEBS) is made up of independent functional components interacting with each other via events [4]. An *event* is a data representation of a happening in the system or the environment in which the system executes.

Events are generated by components called *publishers*. Components interested in the events that have been generated, are called *subscribers*. Other characteristics of DEBSs are:

- The kinds of events, components publish and subscribe to, can be introduced to, and removed from, the system at run time.
- A component publishes an event by explicitly invoking an `announce` or `publish` operation.
- Components must register their interest on the events they want to be notified about.
- When an event is published, the component announcing the event continues its execution without being blocked. The publisher component does not wait for subscriber components to be notified or for their reaction to the event.
- DEBSs maintain the information required to decide which components are to be notified when an event is published, without the need for publisher components to be aware of which components are interested in their events. Hence, when an event is announced the publisher of the event does not specify the components that will be notified of the event.
- The system attempts to deliver an event to all the components interested in the event.

Because of these characteristics, components in DEBSs exhibit time, space, and synchronization decoupling [7]. Time decoupling occurs when interacting components do not need to be actively participating in the interaction at the same time; space decoupling occurs when components do not need to know each other for them to interact; and synchronization decoupling occurs when publishers announcing events do not wait for the events to reach interested components, nor for their reaction to the events.

1.2 Context-Aware Systems

Context can be defined as any information used to characterize the situation a system entity or user is inserted[6].

A context-aware system is a system that takes advantage of available context information relevant to its functionality in order to better assist its users. Context-aware systems are dependent on many different types of context that can either be automatically acquired through sensors, or that is manually input into the system. The former is referred to as implicit context, while the later is referred to as explicit context.

Context-aware systems may be classified according to its main goals: *information and services*, *automation* or *context information repository*[6].

Information and services context-aware systems provide users with extra context information to support the their tasks and goals such as assisting in finding a Japanese restaurant close to the user, for example. Automation uses context information to automate tasks such as switching a mobile phone to silent mode during a scheduled meeting time. Context information repository is used to store context information for later retrieval, such as providing the location where a handheld detected a wireless hotspot.

Requirements for dealing with context-awareness have been discussed in [2, 6] and are described below.

Separation of concerns addresses the need of separating low-level programming details of retrieving context information sensed through sensors from higher-level details such as how this context information applies to the context-aware system. This problem is best addressed through the use of common interfaces for retrieving context information for sensor components.

Distributed communications specifies the need of supporting distributed components and context sources. Context sensing is a distributed effort, since most devices will have sensors for all different types of context.

Context availability relates to the necessity that context-aware systems have of always being able to access context information. Context sensor components should be self-contained and independent from

other components in the system.

Resource discovery relates to the ability of a component being able to identify other components of interest in the system. It addresses the need of being able to request context from other components that either sense, interpret or transform context information.

This report is structured as follows. Section 2 compares our approach to other approaches proposed in the literature. Section 3 describes our metamodel in detail. The applicability of the approach is illustrated in Section 4 by showing how the existing context-aware systems can be modeled using our metamodel. Conclusion and future work are presented in Section 5.

2 Related Work

Different approaches for dealing with context-aware systems have been discussed in the literature. The Context-Toolkit[6], one of the first context-aware frameworks, proposes the use of context widgets as an abstraction for context sensors, hiding the complexity and low level details of context acquisition. Widgets are reusable entities and can be exchanged by other widgets that provide the same type of context. Context can be retrieved from widgets by either polling the widget or requesting for notification on context changes.

The Service-Oriented Context-Aware Middleware (SOCAM)[11] is a distributed middleware supporting context-aware services. Context is provided by either direct sensory input through context providers or by processing and logic reasoning from previously acquired context through context interpreters. The service locating service is a directory containing the references to context providers and context interpreters available for use. It is responsible for keeping track of the changes in context providers and context interpreters. Context-aware services query the service locating service in order to receive a reference from either a context provider or a context interpreter that provides the desired context information. A context-aware service can either pool or request notification upon context changes from the context providers and context interpreters.

A similar approach is presented by *de Farias et al.*[5]. Both context providers and context interpreters are used in a way similar to SOCAM. It introduces a service provider component that provides both context-aware and plain services. A service manager is responsible for discovery, selection and publication of services, while a monitor manages application service subscriptions. A metamodel for modeling Service-Oriented context-aware applications is also provided in their approach.

The Context-Awareness Sub Structure (CASS)[8] is a middleware-based approach that supports context-awareness in hand-held and mobile computing devices. Context is acquired by sensor equipped computers called sensor nodes and transmitted to the CASS server for storage. The CASS server stores all context changes in a database, creating a context history. An inference engine uses the information contained in the context history database to derive higher level context for the context-aware applications. The context-aware applications retrieve context information either through polling or notification. This approach provides a separation between context reasoning and application code, since all of the context processing is contained in the CASS server.

The approaches described above, although referring to networked and distributed systems, are not applicable as a DEBS, since they make use of a Service-Oriented architecture instead. These approaches also focus mostly on low-level programming infrastructure and implementation details of context-aware applications and not on higher-level concepts such as metamodels. The work of *de Farias et al.* is an exception to this, but it also uses a Service-Oriented architecture and does not provide a distributed event-based metamodel for context-aware systems.

The feasibility of applying context-awareness to distributed event-based applications was observed by *Biegel et al.* through the sentient object model presented in [2]. Their proposed approach is based on STEAM[12], a location-aware, event-based middleware for *ad-hoc* networks. In this approach sensors are used to propagate different types of events containing contextual information. Actuators register for specific event types in order to receive context data. Events can be filtered by its context data and location. This approach also focuses mostly on low-level programming infrastructure and does not provide a formal metamodel for context-aware DEBS.

3 Metamodel

We represent DEBSs using our metamodel proposed in [4]. The main advantage of using this metamodel is its ability to represent, not only basic features common to most DEBSs, but also advanced DEBS features that have been lately proposed in DEBS. In particular, component grouping mechanisms and basic role based access controls [1, 10]. Besides the events and component themselves, other DEBS entities represented in the metamodel are *reactive component interfaces*, *regions*, *event schemas* and *access roles*.

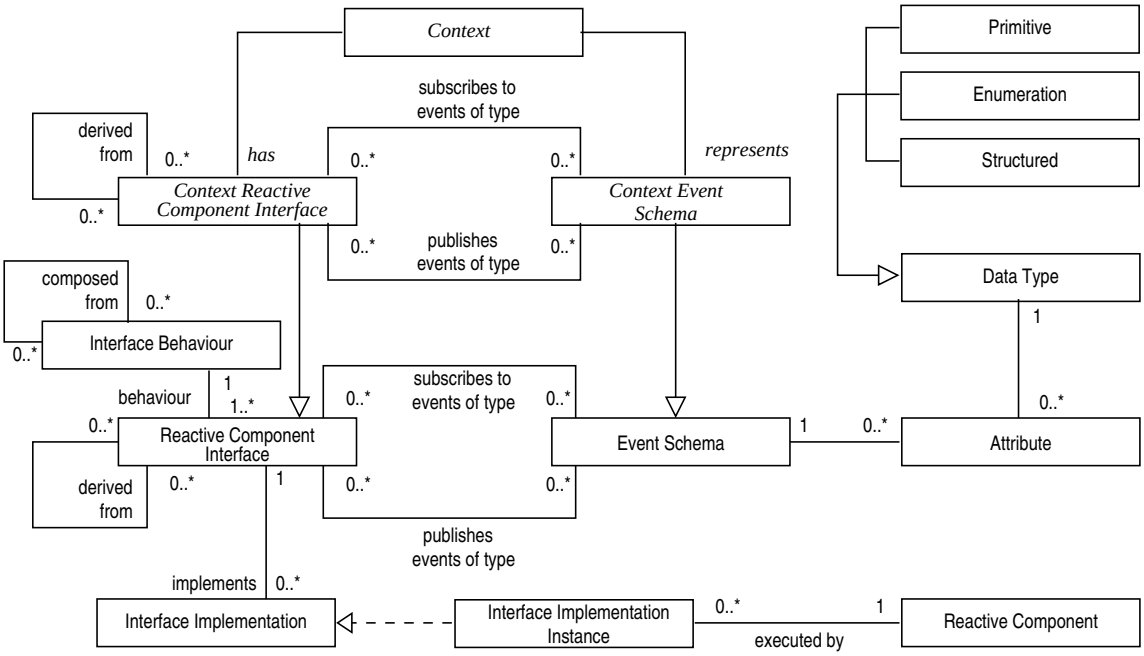

Figure 1: Structural view of the context-aware DEBS metamodel.

The metamodel, as presented in [4], does not provide any context or context-awareness modeling capabilities. Modeling context-awareness with such metamodel would require both context and context-awareness to be modeled at application-level, resulting once again in a low-level programming infrastructure approach.

In order to model context-awareness in DEBSs we need to introduce the concept of context and context-awareness into the existing model. This is accomplished with the inclusion of three new entities into the metamodel: *Context*, *Context Event Schema* and *Context Reactive Component Interface*.

3.1 Structural View

Figure 1 shows the structural view of the context-aware DEBS metamodel. The entities named in *italics* in the Figure are the ones introduced in our approach. These new entities allow for context and context-awareness to be represented as first class citizens that can be modeled and reasoned upon in the metamodel.

Every event in the system has an event schema. An event schema specifies the data attributes that every event of the given event schema must have.

The operation of the components in the system is specified by reactive component interfaces. These interfaces specify the events published and of interest to components, as well as the behaviour that a component implementing an interface must exhibit. The use of interfaces makes possible to focus on what functionality is being provided and the interactions and collaborations between components, without the need to inspect actual interface implementations.

Access roles characterize sets of components [9]. Before publishing an event, a component must advertise the event to the system. Advertisement of an event is rejected by the system if the component executing the request has not been granted the access role required to implement the interface.

Once an event type has been advertised, a component can publish the event by invoking an event publishing operation. A component wishing to react to events of a schema published via a specific interface must first subscribe to the events generated by the interface. A component can subscribe to an event schema produced by an interface, only if it has been granted the role required by the interface to react to events of the given event schema. As part of the subscription request, a component can specify a filtering condition. Events will be delivered to the component only if the filtering condition is met.

The *Context* entity is used to represent a specific type of context. A *Context Event Schema* extends the event schema and is used to represent this context. An event based on a context event schema will be used to propagate the changes in context through-

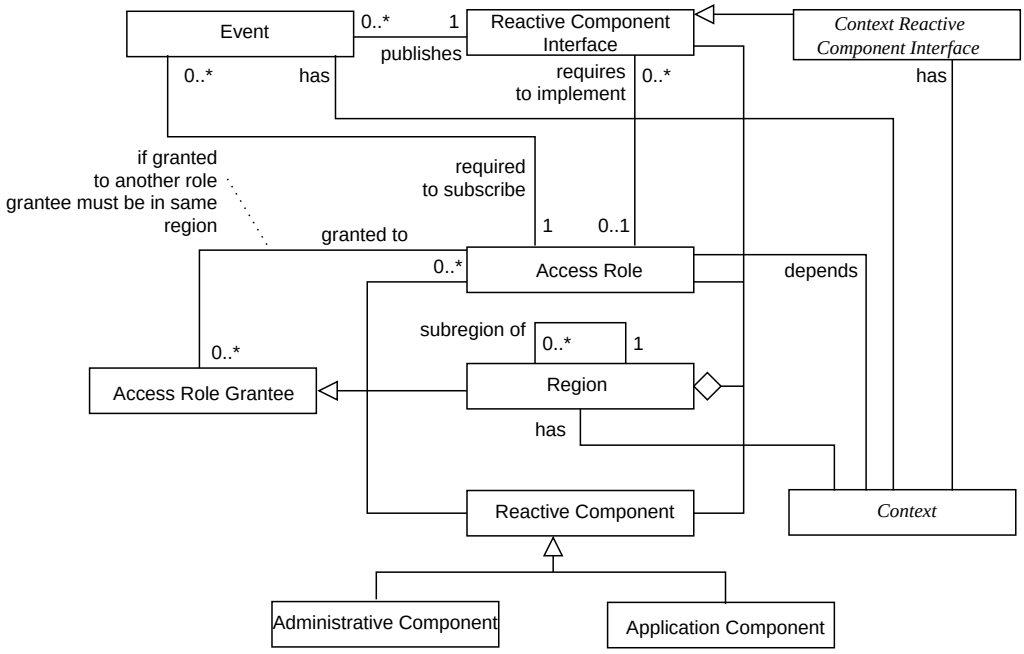

Figure 2: Control view of the context-aware DEBS metamodel.

out the system. It has to provide all of the information described by the context event schema that represents the context being sensed. Context event schemas are used both in the subscription and publication of events.

The *Context Reactive Component Interface* extends the reactive component interface, adding behaviour for generating and reacting to context events. A context reactive component interface is context-aware in that it either senses changes in context (a sensor component in this case) or reacts to context events. This reaction can vary from a simple operation on context change, a simple reactive component, to more complex behaviour such as a context interpretation that infers higher-level context information by combining different context events and propagates it through the system. This allows us to model the behaviour of all common approaches for context-aware systems.

Notification of context events is accomplished through subscription to the context event schema describing the desired context event. The context reactive component interface can also filter context events by the context data described in the event, in a way similar to current DEBSs.

3.2 Control View

Figure 2 shows the control view of the context-aware DEBS metamodel. The *Event* entity represents a happening in the system, and can be categorized according to its schema. Event schemas are stored and maintained by specialized administrative components in each region, and the name of an event schema is assumed to be unique.

A context event will always follow a context event schema. A context event is used to dynamically propagate context information throughout the system as context changes occur. It is also interesting to note that events themselves, being part of the context-aware system, have context associated with them. The most basic context information associated with an event are the component of origin and the timestamp for the publication of the event.

Two kinds of components exist in DEBSs: application and administrative components. Application components run application-specific interface implementations that are not related to the basic operation of the DEBS and its services. Administrative components, on the other hand, are in charge of the administrative activities in the DEBS.

Components in DEBSs are logically grouped in regions. Regions can contain other regions forming a tree hierarchy. With the exception of the root region, regions can be dynamically created and removed from the system.

Basic access control is modeled by using roles. As previously mentioned, a role typically characterizes a set of components. Roles are used to restrict the col-

lection of components that can run given interface implementations, or that can subscribe to certain events. Roles are uniquely identified and can be dynamically created and dropped. Roles are granted to (revoked from) components, regions, and other roles. When a role is granted to a region, every single component in the region is granted the role. Similarly, if a role is revoked from a region, every component in the region is revoked the role, even if the role was directly granted to a component in the region. As with event schemas, role names are assumed to be unique.

In a context-aware system, regions serve the purpose of allowing the grouping of components that share the same context. This permits the modeling of different regions with different contexts. A subregion will contain only a subset of the context that its super-region has available. Overlapping regions will only have part of their context in common. An example of this capability is modeling a context-aware system where the system contains components interested in the context shared with your peers in a social network such as preferences and social relationships, while other components are interested in physical attributes of the context such as room temperature, time and location, while yet other components are monitoring the user's health indices such as glucose levels and heart rate.

Context also has an impact in role access control. Access roles can be granted or revoked based both on context and on context changes. This allows a more fine grained solution in security and privacy for context-aware systems.

4 Model Evaluation

In this section we illustrate the applicability of our metamodel by modeling different context-aware systems proposed in the literature.

4.1 Context-Toolkit

Figure 3 shows the modeling of the Context-Toolkit [6]. Table 1 describes the different components present in the Context-Toolkit framework.

A *widget* is a common interface for accessing context information. It serves the purpose of hiding the complexity of retrieving sensed context information from the sensors; it abstracts context information and provides reusable and customizable building blocks for context-aware applications. In our model, widgets are represented as context reactive component interfaces. All of the knowledge needed to extract sensed context information from sensors is stored inside the interface behaviour. It advertises a context event schema with the structure of the context being provided and publishes events upon context changes. This context reactive component interface can be reused by other components in the system. Events published by one interface can be combined with events published by other interfaces to provide customization capabilities.

Component	Description
Aggregator	logical repository of context information
Discoverer	component registry
Interpreter	context information transformer
Sensor	a provider of context information
Service	an external action
Widget	common interface for context retrieval

Table 1: Context-Toolkit framework components.

An *aggregator* is a logic repository for context information. The aggregator serves two different purposes, the first being a central location that stores all of the context information pertaining to an entity in the system, and the second being a central location that stores the same context information form different context sources. In our approach, the first use of aggregators is not needed. Once a component subscribes to a context event schema it is guaranteed to receive events that are based on that schema. In this view, the component itself if the aggregator of context information, without any need for an external aggregator. Examples of this use of aggregators are shown in the "Intercomm System" and the "Conference Assistant System" presented by *Dey et al.* in [6].

The second use of aggregators relates to our use of context event schemas, since it deals with accessing the same context information from different sources. If a component subscribes to a given context event schema it will receive events from all components that use that schema for event publication. Examples of this use of aggregators are shown in [6] through the model implementation of the "Active Badge System" and the "Mobile Tour Guide".

A *discoverer* is a component responsible for providing a registry of all widgets, interpreter and aggregators available in the system. It is implemented in our model through an administrative component which is responsible for keeping the registry of advertised event schemas in the system.

An *interpreter* is a component that transforms low-level context information into higher-level context information through reasoning and inference. This

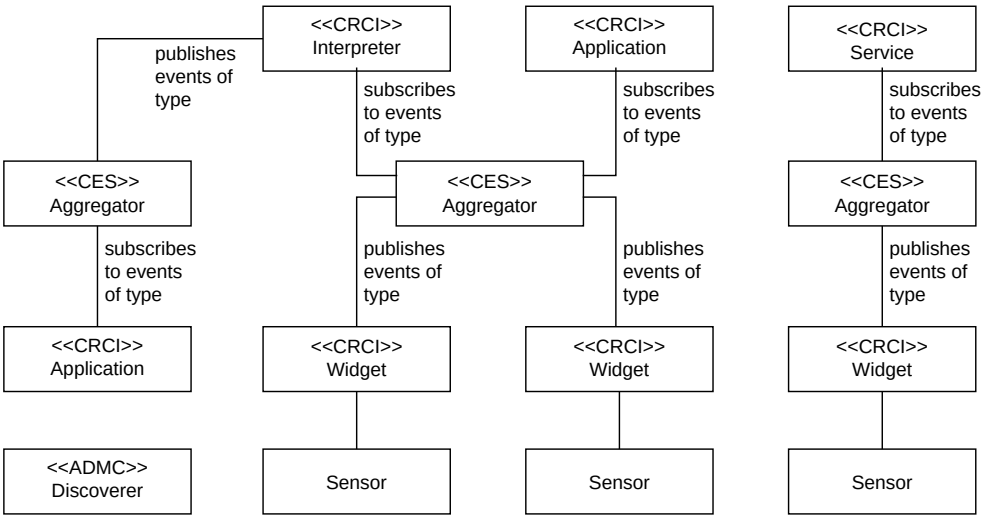

Figure 3: Modeling the Context-Toolkit framework.

component is modeled in our approach as a context reactive component interface. This interface will have the associated behaviour of subscribing to the relevant context event schemas and processing the received events to extract higher-level context information. Once this new context is derived from subscribed events, the component will publish a new event with this higher-level context information. This new event should be based on different context event schema in order to identify the different in context information.

A *sensor* is simply a source of context. In the Context-Toolkit framework sensors are entities external to widgets, in our model we present them as being external entities to the widgets, but both the sensor and the widget can be part of the same component.

Component	Description
Context Provider	context information interface
Context Interpreter	transforms context information
Context Database	context history and storage
Context-aware Service	context-aware application
Service Locating Service	component registry
Sensor	source of context information

Table 2: SOCAM components.

4.2 SOCAM

Figure 4 shows the modeling of SOCAM[11]. Table 2 describes the components of the SOCAM framework.

Sensors are components able to sense a given type of context. SOCAM defines two different types of sensors: virtual sensors and physical sensors. Virtual sensors, web services or information servers for example, provide some context information that is external to the device containing the context-aware application. Physical sensors, on the other hand, are ubiquitous sensors associated with the environment where the device containing the context-aware application is present. We model sensors as context reactive component interfaces. This allows a sensor to specify the behaviour of publishing events of a certain context event schema when a change in context is sensed. The low-level programming concepts for context acquisition are also hidden inside this component.

Context Providers are components that serve as an interface for context acquisition in sensors, being able to separate the low-level programming concepts relating to context sensing from higher-level concepts of context-aware applications. SOCAM differentiates between external context providers and internal context providers, with the former relating to external sensors and the later relating to physical sensors present in the context-aware domain. Context providers are modeled as context event schemas, since its objective is to provide context information. Context providers act as interfaces for context information acquired from sensors in the same way that context event schemas act as context information pub-

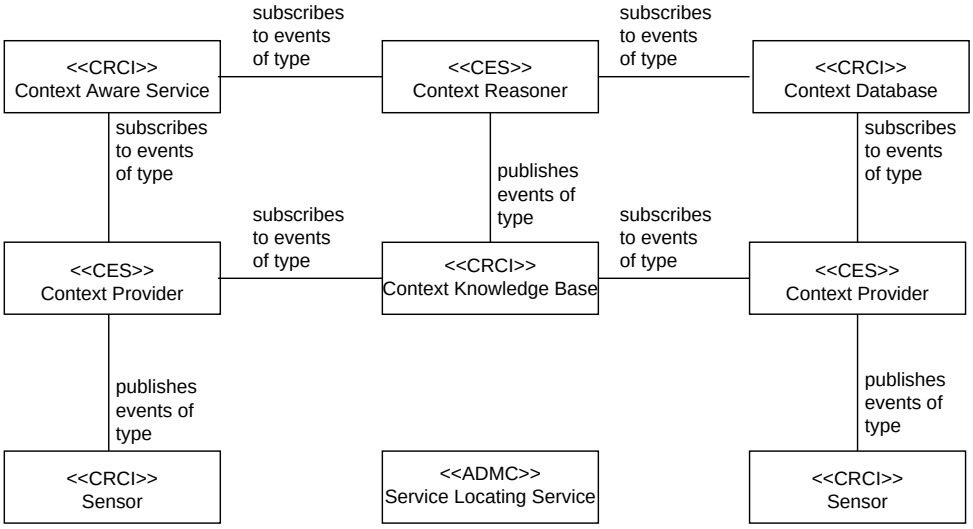

Figure 4: Modeling SOCAM.

lished by a context reactive component interface. In our model we do not differentiate between external and internal context providers for the sake of simplifying the model. The functionality for both internal and external context providers is the same, though, and as such this differentiation can be modeled with ease using our approach.

Context Interpreters are specialized components composed of a *Context Knowledge Base* and a *Context Reasoner*. A context interpreter uses logic reasoning to process context information. This processing involves extracting higher-level context information, querying context knowledge, ensuring consistency and resolving conflicts in context information. The context reasoner is responsible for the logic reasoning and processing, while the context knowledge base provides an API for adding, querying, modifying and deleting context information. In our model we show both the context knowledge base and the context reasoner components explicitly instead of abstracting both components inside a context interpreter.

The addition, deletion and modification of context information is done through the context knowledge base which is modeled as a context reactive component interface. Adding a new context provider and new context information can be accomplished by subscribing to a previously unsubscribed advertised context event schema. Since events are published on context changes, the context information is updated whenever a new event is received, keeping the context knowledge base up to date. The deletion of context information from the context knowledge base occurs when the last context provider publishing events of a give context event schema removes (unadvertises) this event schema from the system.

The logic reasoning is also performed through the context knowledge base. It receives context information from the context providers and processes this information to produce a new, higher-level context. Similarly to context providers, this higher-level context information is published through a context event schema, and propagated throughout the system. This context event schema is the context reasoner component.

Context-aware services are services that use the context information from context providers and context interpreters. They are modeled as context reactive component interfaces, and subscribe to the context event schemas providing the context information they require.

Context databases are repositories of context history for the domain. They hold all of the past context information received from context providers and context interpreters. They are modeled as context reactive component interfaces, and subscribe to all context event schemas available in the domain.

Service locating services provide a registry for all context sources present in the system. It is modeled as an administrative component that contains all the advertised context event schemas advertised by the sensors and by the context interpreter, represented by knowledge base. These context event schemas are the context provider and the context reasoner components.

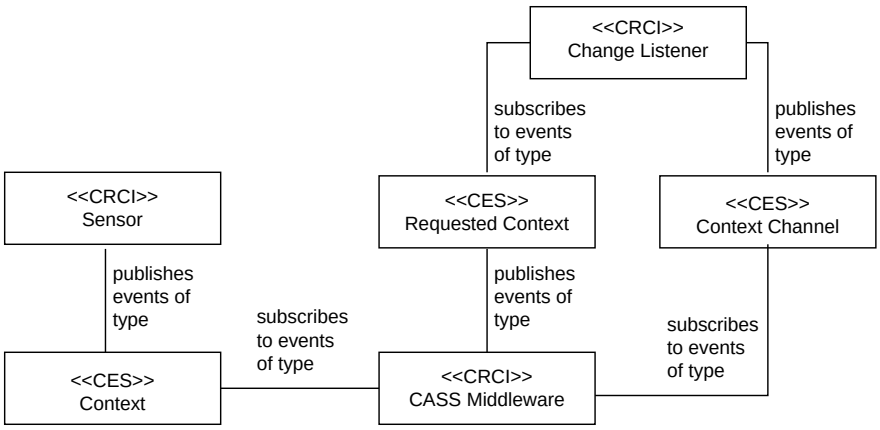

Figure 5: Modeling CASS.

4.3 CASS

Figure 5 shows the modeling of CASS[8]. Table 3 describes the components in the CASS framework.

Component	Description
Cass Middleware	middleware components such as interpreter and context retriever
Change Listener	mobile listener for context changes
Requested Context	requested context information
Sensor	source of context information
Sensor Listener	listener for context changes

Table 3: CASS components.

Context is represented by a context event schema. There is a context event schema for each different type of context. A *sensor* is responsible for acquiring context information. Sensors are modeled as context reactive component interfaces that advertise the context event schema that refers to the context it is able to sense. Sensors sensing the same type of context share the same context event schema. Sensors publish context events following the context event schema whenever a change in context is sensed.

Change listeners are the context-aware components of the system. Change listeners are modeled as context reactive component interfaces. These components have the ability to receive the desired context information by subscribing to a context event schema of type requested context. Upon subscription, the change listener defines a filtering condition. This filtering condition specifies that only context events containing context information requested by this change listener be delivered. The request of context information is accomplished through the advertisement and publication of an event of type *context channel*.

The *CASS middleware* component is composed of a context database as well as two subcomponents: *context retrievers* and *interpreters*. The middleware is responsible for gathering all context information available; it is modeled as a context reactive component interface that subscribes to all context event schemas advertised by the sensors. It stores context information in its context database, and this context information can be retrieved by change listeners at a later time.

Context retrievers are components that query the context information database and retrieve the desired context information, according to context channel events. The retrieval of past context information is accomplished by subscribing to the context channel. A change listener publishes an event of type context channel with a request for some specific context information, stored by the sensor listener. The sensor listener receives this event and publishes an event of type requested context with the context information being requested. The filtering condition ensures that the event will be received by the change listener that requested the context.

Interpreters are transformers of context information, being able to extract higher-level context information from available context. Upon receiving a request for a higher-level context information, the middleware will invoke the interpreter and retrieve the necessary context information from the database for logic reasoning processing. The higher-level context information is the result of this processing and is published by the middleware as an event of type requested

context. Once again, the filtering condition ensures that the event will be received by the change listener that requested the context information.

5 Conclusion and Future Work

In this report we addressed the lack of higher-level modeling abstractions for context-aware systems. Our approach focuses on a metamodel that supports context-awareness features and modeling capabilities, as well as dynamic event-based interactions. This metamodel addresses the requirements dealing with context-awareness listed in Section 1.2. Separation of concerns is addressed by the use of reactive component interfaces. The event-based infrastructure deals with distributed communication while the publish and subscribe model targets the context availability requirements. Finally, resource discovery is handled through schema advertisement. We illustrate the applicability and validate our metamodel by representing existing context-aware systems.

Future work includes continuing to extend DEBS concepts such as access roles to deal with both privacy and context-aware security issues, and a context-aware publish and subscribe model. Work is also being done in the formalization of the context-aware DEBS computational model by extending kell-m [3], and in providing property validation mechanisms. A context metamodel is being developed that will enable the modeling of different types of context.

References

[1] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based Access Control for Publish/Subscribe Middleware Architectures. In *2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, San Diego, California, June 2003. ACM. Program Chair-Hans-Arno Jacobsen.

[2] G. Biegel and V. Cahill. A Framework for Developing mobile, Context-Aware Applications. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 361, Washington, DC, USA, 2004. IEEE Computer Society.

[3] R. Blanco. *Process Models for Distributed Event-Based Systems*. PhD thesis, Univeristy of Waterloo, Ontario, Canada, March 2010.

[4] R. Blanco, J. Wang, and P. Alencar. A Metamodel for Distributed Event Based Systems. In R. Baldoni, editor, *Proceedings of the Second International Conference on Distributed Event-Based Systems*, volume 332 of *ACM International Conference Proceeding Series*, pages 221–232, New York, NY, USA, July 2008. ACM Press.

[5] C. R. G. de Farias, M. M. Leite, C. Z. C. R. M. Pessoa, and J. G. P. Filho. A MOF Metamodel for the Development of Context-Aware Mobile Applications. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 947–952, New York, NY, USA, 2007. ACM.

[6] A. K. Dey, G. D. Abowd, and D. Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human Computer Interactions*, 16(2):97–166, 2001.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[8] P. Fahy and S. Clarke. Cass - Middleware for Mobile Context-Aware Applications. In *Proceedings of MobiSys Workshop on Context Awareness*, pages 304–308, June 2004.

[9] D. Ferraiolo and R. Kuhn. Role-based Access Controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[10] L. Fiege. *Visibility in Event-Based Systems*. Ph.d. thesis, Technische Universität Darmstadt, Darmstadt, Germany, Apr. 2005.

[11] T. Gu, H. K. Pung, and D. Q. Zhang. A Service-Oriented Middleware for Building Context-Aware Services. *Journal of Network and Computer Applications*, 28(1):1–18, 2005.

[12] R. Meier and V. Cahill. Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications. In *In Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*, volume 2893 of *Lecture Notes in Computer Science*, pages 285–296. Springer-Verlag, 2003.